ECE 595 Reinforcement Learning: Coding Project

John Tran, Lucas Zhou

I. Introduction

THIS report will go over the two coding projects: `Find Me If You Can!` and `Baby Bella's Quest: Optimal Paws-itioning in a Grid World`. A brief overview of each coding project will be given as well as any assumptions made when implementing them. There will also be four subsections going over further details of each coding project: Assumptions (Extension of Problem), Special Implementation Considerations, Results of Code, and Discussion of Observations.

As for the implementation details, they will be reserved for the presentation video (high level overview) and the comments in the code will serve as a better resource than notes given in this report.

II. Find Me If You Can!

The first coding assignment asked us to implement a bandit algorithm on an application of a Positioning, Navigation, and Timing (PNT) system. From the list of five anchor nodes (anchor positions) given to us, we are asked to choose three anchor nodes (taken collectively as a single action) and update the position estimate on a given target. This is also known as the trilateration technique.

The results obtained by strictly following the details specified required additional adjustments to produce the "expected" results mentioned in class (i.e., final euclidean distance was about 5.6). The assumptions made will be discussed in the following section.

A. Assumptions (Extension of Problem)

First, the three functions f, g, h defined was modified somewhat to provide clearer details during implementation. The original expression of f is given below:

$$f\left(\hat{x}_t^{(ite)}, \hat{y}_t^{(ite)}, \hat{z}_t^{(ite)}\right) = \sqrt{\left(x_a - \hat{x}_t^{(ite)}\right)^2 + \left(y_a - \hat{y}_t^{(ite)}\right)^2 + \left(z_a - \hat{z}_t^{(ite)}\right)^2} - \hat{d}_{a,t}$$

where

$$\hat{d}_{a,t} = \sqrt{\left(x_a - \hat{x}_t^{(ite)}\right)^2 + \left(y_a - \hat{y}_t^{(ite)}\right)^2 + \left(z_a - \hat{z}_t^{(ite)}\right)^2}$$

However, the differentiation between $\hat{d}_{a,t}$ and the first term on the right of f was not clear from reading the given problem. So, after some clarification on how to calculate the Jacobian, we used the fact that $\hat{d}_{a,t}$ was treated as a constant (i.e., vanished when taking the partial derivatives of f) to slightly modify the definition of f (also applicable for g, h):

$$f\left(\hat{x}_t^{(ite+1)}, \hat{y}_t^{(ite+1)}, \hat{z}_t^{(ite+1)}\right) = \sqrt{\left(x_a - \hat{x}_t^{(ite+1)}\right)^2 + \left(y_a - \hat{y}_t^{(ite+1)}\right)^2 + \left(z_a - \hat{z}_t^{(ite+1)}\right)^2} - \hat{d}_{a,t}$$

The important change is the time step change on all the terms ***except*** $\hat{d}_{a,t}$ from ite to $ite+1$. This is consistent with how $\hat{d}_{a,t}$ is treated as a constant it involved terms with the previous time step ite (i.e., $\hat{d}_{a,t}$) and not terms with the current time step $ite+1$ which the partial derivatives should be using. For example, the partial of f with respect to x would be

$$\frac{\partial f\left(\hat{\boldsymbol{p}}_t^{(ite+1)}\right)}{\partial \hat{x}_t^{(ite+1)}}$$

where the denominator of the partial derivative has the explicit hat $\hat{x}$ and the specific time step to make it clear which time step is involved in the partial derivative. This would apply with respect to y, z and the other two functions g, h as well.

So, we have the three functions involving terms from both $ite+1$ and ite times steps (where ite terms are used indirectly from $\hat{d}_{a,t}$). To see how the first iteration is handled (i.e., only the initial time step is defined with no previous time step), please see the code for more details – basically, the unit vector from the initial position estimate to the target position (scaled appropriately) is used to derive a temporary previous position estimate along the reverse direction of the unit vector, and its only purpose is to help calculate the Jacobian during the first time step with our modifications to the three functions outlined above.

Then, we added a scaling α value associated with the $\Delta = (\Delta\hat{x}_t, \Delta\hat{y}_t, \Delta\hat{z}_t) \rightarrow \alpha(\Delta\hat{x}_t, \Delta\hat{y}_t, \Delta\hat{z}_t)$ in updating the position estimate. For example, in updating the x component of $\hat{\boldsymbol{p}}_t^{(ite+1)}$, the α is added in front of the $\Delta\hat{x}_t$: $\hat{x}_t^{(ite)} + \alpha\Delta\hat{x}_t$ (applies to the y, z components as well). This separate α value unique to the Δ was added to accommodate the changes made to the three functions mentioned previously – the new Δ would result in large changes to the position estimate and lead to inconsistent results. So, the α value should be less than 1 to reduce the changes made in between time steps.

Also, a dynamic version was done on the α value for Δ to make the distance error more stable during the later half of the bandit algorithm (more sensitive to changes). It would take the current time step as an argument and calculate the α value and scale it based on an interval between two specified values (i.e., start and end value) and the time step would slowly reduce the impact of the Δ on the position estimate as the run of the program progressed. This still produced varying results from run to run which will be discussed more in the Results of Code section.

B. Special Implementation Considerations

When looking at the given definition of the Jacobian $J^{(ite)}$, the partial derivatives of the three functions f, g, h are ordered row-wise (i.e., partial derivatives of f along the first row, g along the second row, and h along the third row). However, the assignment of the three anchor nodes to these three functions in a given action was left unclear. Luckily, we can show that the mapping of anchor node to function does not matter with a little bit of linear algebra.

First, we find two important spots where the Jacobian is used: calculating $GDOP(A)$ for an action A (composed of the three chosen anchor nodes) and calculating $\Delta = \alpha(\Delta\hat{x}_t, \Delta\hat{y}_t, \Delta\hat{z}_t)$. From here, we can look at each case and show that the reordering of the rows of the Jacobian will not change the resulting answer in each.

For simplicity, we can label the five anchor nodes as a_0, a_1, a_2, a_3, a_4 where the subscript can stand for the index which a anchor node is located in the given list. Also, for a given action A and the three corresponding anchor nodes, the three functions will be mapped according to increasing index (i.e., for chosen anchor nodes a_i, a_j, a_k where $i < j < k$, f will be mapped to anchor node a_i, g to a_j, and h to a_k). This will keep the discussion of the setup simple and keep the notation consistent.

From here, we can define $J_{ijk}^{(ite)}$ to be the Jacobian with the mappings defined above (f, g, h to anchor indices i, j, k in that order). So if we consider an example where the mappings were different, like $J_{jik}^{(ite)}$, then f, g, h would be mapped to anchor indices j, i, k instead. Since the only difference among the three functions is the associated anchor node, the mapping could be reworded as g, f, h mapped to anchor indices i, j, k – easier to visualize when dealing with row swaps. However, we can write $J_{jik}^{(ite)}$ as an expression of $J_{ijk}^{(ite)}$ through some row swaps – in this case, swapping the first and second rows. This can be written simply with a permutation matrix P in front of $J_{ijk}^{(ite)}$ where

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and

$$J_{jik}^{(ite)} = PJ_{ijk}^{(ite)}$$

This form can be applied for any mapping of anchor nodes to the three functions (some permutation of the ordered Jacobian $J_{ijk}^{(ite)}$ above). The final tools we need are some properties of P – specifically, noticing that P is an orthogonal matrix where $P^{-1} = P$ and $P^{-1}P = P^TP = I$.

Looking at $GDOP(A)$, we have

$$GDOP(A) = \sqrt{\sum_{\forall i} G(i,i)}$$
$$G = \left(J^{(ite)^T} \cdot J^{(ite)}\right)^{-1}$$

and we need to show that G derived for different permutations of $J_{ijk}^{(ite)}$ ($PJ_{ijk}^{(ite)}$) should be the same.

$$\begin{aligned} G &= \left(\left(PJ_{ijk}^{(ite)}\right)^T \cdot PJ_{ijk}^{(ite)}\right)^{-1} \\ &= \left(J_{ijk}^{(ite)^T} P^T \cdot PJ_{ijk}^{(ite)}\right)^{-1} \\ &= \left(J_{ijk}^{(ite)^T} J_{ijk}^{(ite)}\right)^{-1} \end{aligned}$$

We can see the permutation matrix P reduces to the identity and the result is the same as the ordered Jacobian – any permutation would result in the same $GDOP$ value regardless of the mappings of anchor nodes to the three functions.

Then we can do a similar thing for Δ

$$\begin{aligned} \Delta &= \alpha(\Delta\hat{x}_t, \Delta\hat{y}_t, \Delta\hat{z}_t) \\ &= \alpha\left(J^{(ite)^T} \cdot J^{(ite)}\right)^{-1} J^{(ite)^T} \mathcal{RES}^{(ite)} \end{aligned}$$

where we only need to look at $J^{(ite)^T}\mathcal{RES}^{(ite)}$ since $\left(J^{(ite)^T} \cdot J^{(ite)}\right)^{-1}$ is the same expression found in G above (already showed it produced the same result).

$$\begin{aligned} J^{(ite)^T}\mathcal{RES}^{(ite)} &= \left(PJ_{ijk}^{(ite)}\right)^T \left(P \cdot \mathcal{RES}^{(ite)}\right) \\ &= J_{ijk}^{(ite)^T} P^T P \cdot \mathcal{RES}^{(ite)} \\ &= J_{ijk}^{(ite)^T} \cdot \mathcal{RES}^{(ite)} \end{aligned}$$

where P is added in front of $\mathcal{RES}^{(ite)}$ because of its definition on the ordering of the three functions. So, we have shown that both use cases of the Jacobian would produce the same results regardless of which anchor nodes were mapped to which functions (or how the rows are ordered in the Jacobian).

In terms of implementation, a brief note will be given on how this affects the choice of action at each iteration. Since we established the ordering of anchor nodes in the Jacobian does not matter, the increasing index notation was used ($J_{ijk}^{(ite)}$) to decide how the anchor nodes were placed. This also affected what makes two actions different – since the implementation used this increasing index ordering as a key (in a dictionary) to store the action counts and Q value, any permutation of a given three set of anchor nodes would be seen as the same action (e.g., a_0, a_1, a_2 along with a_1, a_0, a_2 and the other permutations would be counted as the same action). This is important when choosing a random action during exploration (random anchor nodes chosen was given as an ordered list). This assumption of grouping all permutations together was valid because three different anchor nodes are chosen as an action which would mean each set of three nodes would result in the same number of permutations, leading to the same probability for each.

Finally, during experimentation of the program, there were some runs that resulted in exploding Euclidean distance error for the $\epsilon = 0.3$ case. This would lead to the program terminating early with a singular matrix error when finding the inverse of the matrix for calculating $GDOP$ and $Delta$.

So, a simple fix was added to bound the distance error achieved by both ϵ cases. If an iteration would produce a distance error above some specified threshold (our program used 10 as the threshold), then everything would be updated normally except the new position estimate (e.g., update action count and update Q value). The position estimate was programmed to not update when the threshold was reached. However, it did not seem this feature was needed after some tweaking of the parameters (discussed further in the following results and observations sections).

C. Results of Code

This section will contain all the plots generated when running the program for $\epsilon = 0.01, 0.3$ with the modified and added features mentioned in the previous sections. Also, the discussion of these plots will be reserved for the next section (figure captions will provide brief observations and more detailed discussion will follow in the next section).

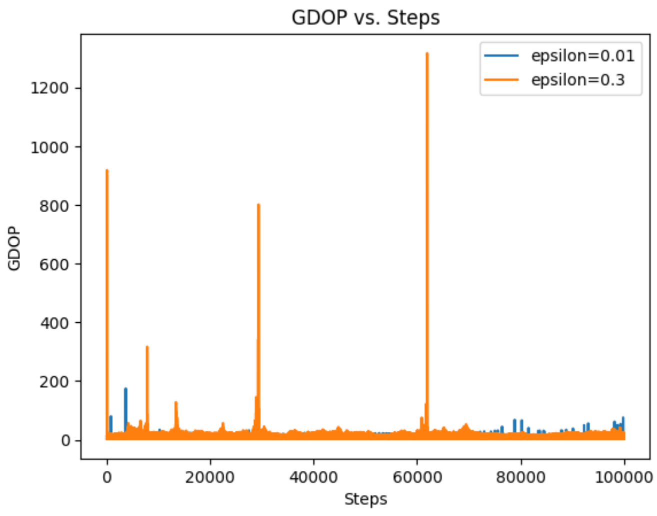

Fig. 1. $GDOP$ vs. time steps of the simple bandit algorithm – large spikes in $GDOP$ is observed for $\epsilon = 0.3$ relative to $\epsilon = 0.01$ which in contrast stays pretty uniform with an overall slight increase towards the end

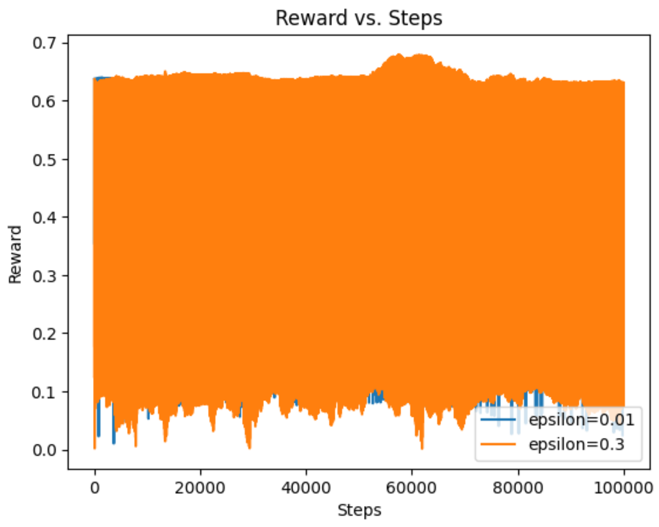

Fig. 2. Reward vs. time steps of the simple bandit algorithm – both ϵ exhibit varying reward values as shown by the block of color (more details with separated plots in the following section)

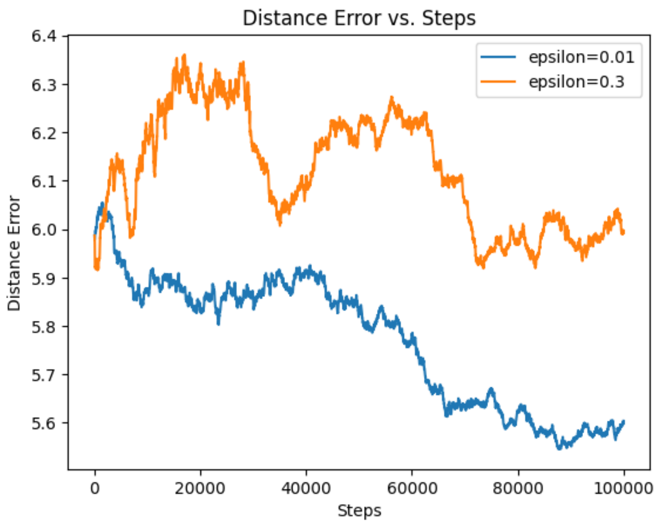

Fig. 3. Distance error (Euclidean distance) of the real and estimated position of target vs. time steps of the simple bandit algorithm – this particular plot gives varying results from different runs but this run gave a nice expected result to discuss in the following section ($\epsilon = 0.01$ provided lower errors for the vast majority of the runtime compared to $\epsilon = 0.3$). Based on the modifications and additions mentioned in previous sections, a dynamic α value for Δ was used that started at 0.95 and gradually (and linearly) decreased to 0.9 at the end of the program run. Bound was set at 10 but it was not needed in this run.

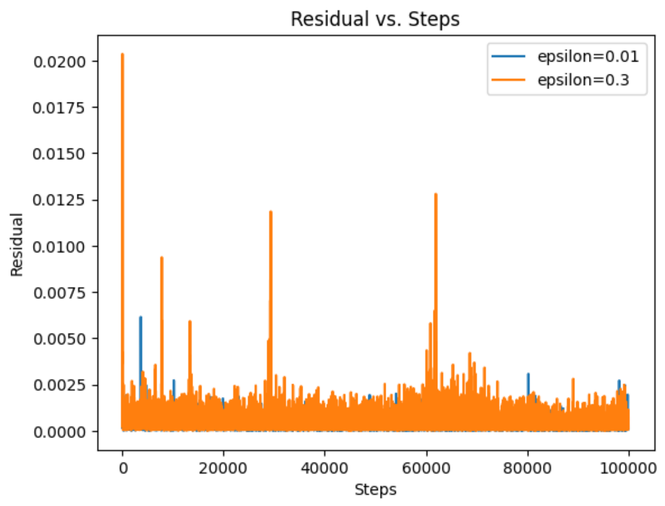

Fig. 4. 2-norm of residual vs. time steps of the simple bandit algorithm – extra plot to help gauge the big jumps seen in the distance error plot for both ϵ (seems similar information is captured with the $GDOP$ plot)

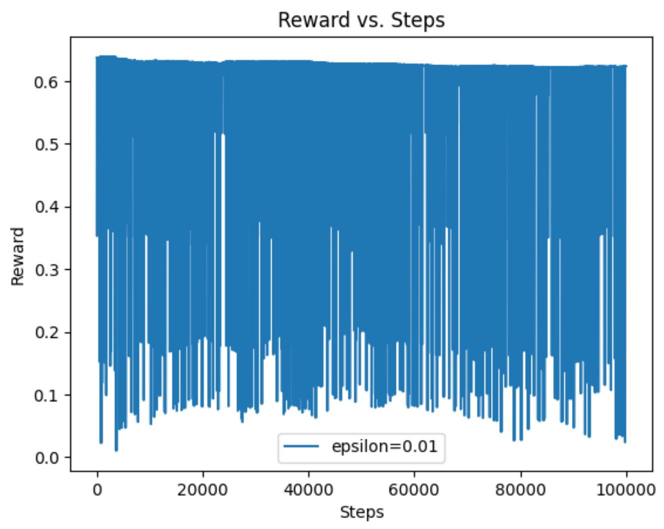

Fig. 5. Reward vs. time steps of the simple bandit algorithm for $\epsilon = 0.01$

D. Discussion of Observations

First for Figure 1, the $GDOP$ values only spiked to relatively high values for $\epsilon = 0.3$. From our understanding, $GDOP$ seems to signal the potential loss of precision as higher $GDOP$ values would mean a lower trace of the Jacobian (and loss of a diagonally dominant matrix and would make finding the inverse harder for the solver due to precision loss). This makes sense since the higher ϵ value would be more likely to take an action that would move it further away from the target due to more exploration. If Figure 3 is referenced with the GDOP values, then it does appear that the big spikes correlate with relatively large jumps in the distance error (either increase or decrease). The Δ to update the position estimate shares the $G = \left(J^{(ite)^T} \cdot J^{(ite)}\right)$ term, which is directly used by $GDOP$ to determine loss of precision (and would result in big jumps). However, looking at Figure 4, the spikes in the residual for $\epsilon = 0.3$ interestingly seem to closely relate with the $GDOP$ values – our guess is that the $GDOP$ value from a previous iteration may influence the residual in the next iteration since the Jacobian was modified to use both the current and previous iteration values and a large Δ (and its components) would certainly affect the modified three functions (difference of the first term on the right of the function, like with f, and $\hat{d}_{a,t}$ is exactly the difference between the previous and current iterations since the residual has the evaluation of the three functions as its components).

Then for the reward, there is clearer explanation when the reward plots for the two ϵ values are separated, shown below:

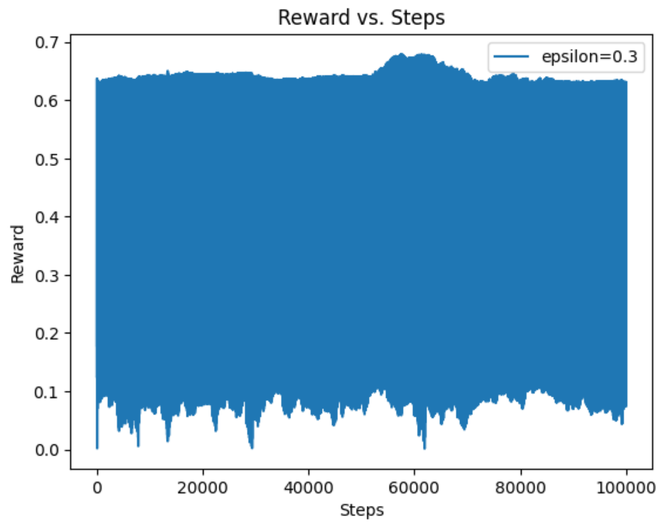

Fig. 6. Reward vs. time steps of the simple bandit algorithm for $\epsilon = 0.3$

Even though both Figures look similar, there is a distinct ”smearing” pattern (reminiscent of rain going down a window) for $\epsilon = 0.01$ (Figure 5) not seen in $\epsilon = 0.3$ (Figure 6), which is just a solid color. Figure 5 has the reward drop to near 0 values infrequently which would make sense since this could indicate time steps where exploration occurred (and produced a bad reward value). Given the large number of time steps ran, the spacing between exploration looks close together but there is clear evidence that the reward remains consistently high for a good portion of the runtime. As for Figure 6, the reward jumps all over the place and the solid color means the reward is varied enough that there is no discernible reward value that the $\epsilon = 0.3$ case has during runtime.

Finally for Figure 3, we were able to achieve the lower limit of distance error mentioned in class (about 5.6) for $\epsilon = 0.01$. There were a few runs where the exploration done by $\epsilon = 0.3$ reached much higher distance errors, making the comparison to $\epsilon = 0.01$ quite hard due to the overall scale of the plot being changed (e.g., upwards of a distance error of 10 for $\epsilon = 0.3$ since the bound was set at 10, as mentioned in previous sections). For the $\epsilon = 0.01$ case, there was less exploration than $\epsilon = 0.3$. The smaller ϵ provided a better path for the agent to not get lost in exploration while gradually getting closer to the target. Even though the higher ϵ had the potential to reach the target faster (and closer too), the distance error turned out to be erratic and only achieved a better distance error at the very beginning of the runtime. Afterwards, it entered two periods of going farther away from the target and then going closer to the target. This could be attributed to the high sensitivity of the PNT system presented to us, even given a particular choice of starting parameters.

III. Baby Bella's Quest: Optimal Paws-itioning in a Grid World

The second coding assignment asked us to implement the value iteration algorithm, which finds the optimal value function and policy, for a grid world environment. We are tasked to create a 2D grid world with movement up, down, left, or right, navigating the grid to a specified terminal state. Empty or blocked cells should also be added to the grid, where the agent must maneuver around.

The grid world is displayed with the blocked cells and the optimal value function for all iterations of the algorithm ending when the grid has converged. In addition, we need to graph the value function changes over all iterations of the value iteration algorithm.

A. *Assumptions (Extension of Problem)*

There were no specific assumptions made since the assignment prompt was pretty open ended with any implementation of the value iteration algorithm in a grid world scenario. Besides implementation and various parameter choices (e.g., reward values for goal and blocked cells, reward step, theta value (threshold for convergence), discount factor (γ)), the main requirements given in the problem statement were completed without extra (necessary) extensions or additions.

B. *Special Implementation Considerations*

Since the assignment prompt was pretty open ended, all necessary parameters were chosen arbitrarily to implement the value iteration algorithm based on the pseudocode from the textbook (Barto and Sutton) that we had discussed theoretically in class.

We chose a grid world size of 5×5 as examples of grid world that we studied were of this size. Though we did do some implementation testing with a 7×7 grid size, the smaller 5×5 was sufficient in clearly showing the value iteration algorithm reach convergence, so we decided to maintain the smaller size. For the smaller grid world, we arbitrarily chose the terminal state to be (0, 4) with blocked cells at (1, 2), (2, 2), (2, 4), and (3, 4). For the larger 7×7 grid world, we chose the terminal state (1, 2) and blocked cells were placed at (3, 3), (5, 4), and (1, 1). All of these grid positions were chosen arbitrarily and modifications to other cell values would yield similar results.

Note: when we say the various block cells and goal state were chosen arbitrarily, we chose some scenario that would lead to fast computation time with an easier time checking the results with what we expected of the program (i.e., the policy produced the right actions at each cell). Also, the 5×5 case was adapted from the screenshot given in the email asking for clarification.

The only exception would be in any instance where blocked cells completely surrounded the terminal state. In any possible cases of this scenario, rewards for any state of the grid world would be negative indicating that it cannot successfully traverse to the terminal state with the up, down, left, right movements.

A negative reward for any action "leaving" the grid world space is unnecessary since the size is always defined, whether it's hard coded like our implementation or user defined, due to the inability for any elements outside a defined array to be chosen. The reward of the terminal state was set at +10 to easily observe changes of the value function after each iteration of the algorithm. The blocked cells were given a reward of 0 to avoid those cells. The step reward was set to -1 to deter from converging on any states that were not the determined terminal state. The discount factor was set to $\gamma = 0.9$ to give more weight to current states as compared to future states. For the value iteration algorithm to converge, the discount factor must be any value such that $\gamma \leq 1$. We also defined θ ($\theta > 1$), the threshold for determining the accuracy of the estimation, to $\theta = 1 \times 10^{-10}$. Considering these arbitrary parameters of our implementation of the value iteration algorithm in a grid world scenario we can further discuss the following results and observations.

C. *Results of Code*

This section will contain the graphical displays of the grid world with the specified terminal state of the 5x5 grid world as well as its respective blocked cells. Each iteration will be shown as well as the grid world with the final rewards for each state in the value function when converged as well as the actions of each state in the value function once converged.

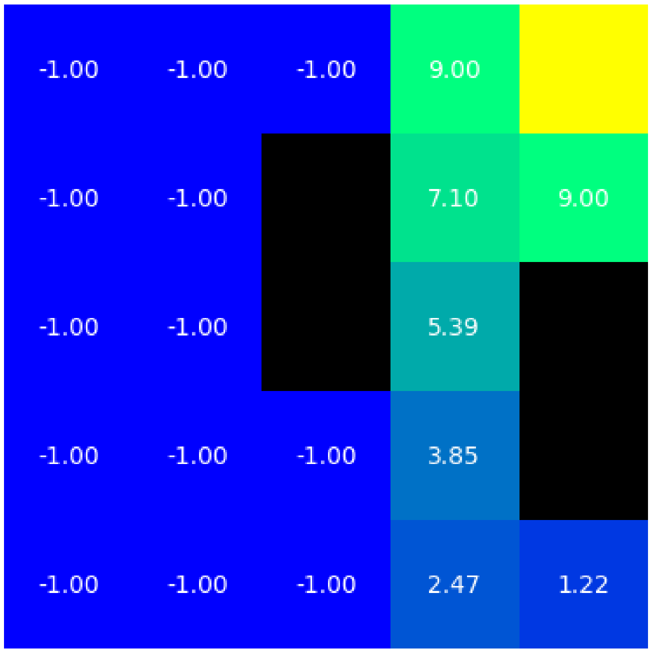

Fig. 7. Grid World after iteration 1

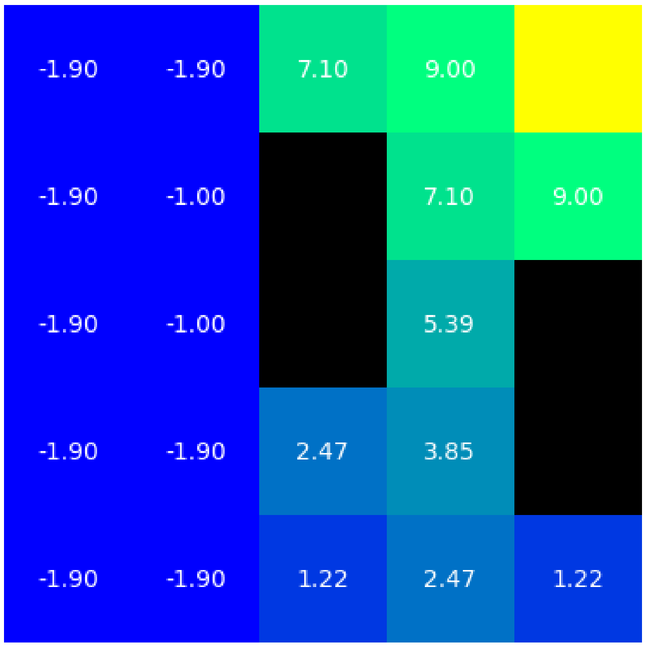

Fig. 8. Grid World after iteration 2

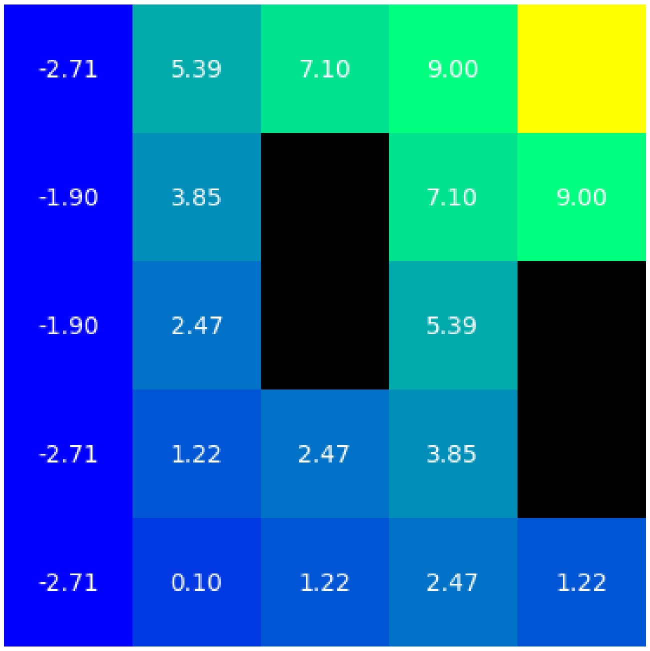

Fig. 9. Grid World after iteration 3

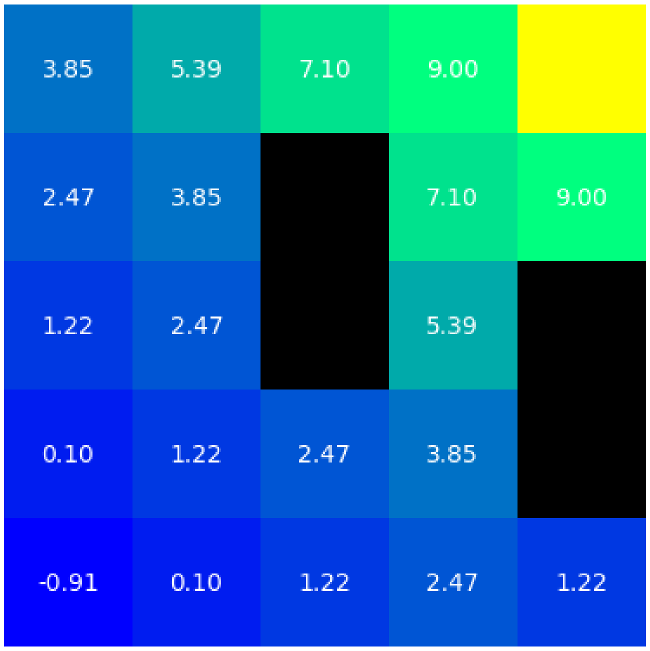

Fig. 10. Grid World after iteration 4

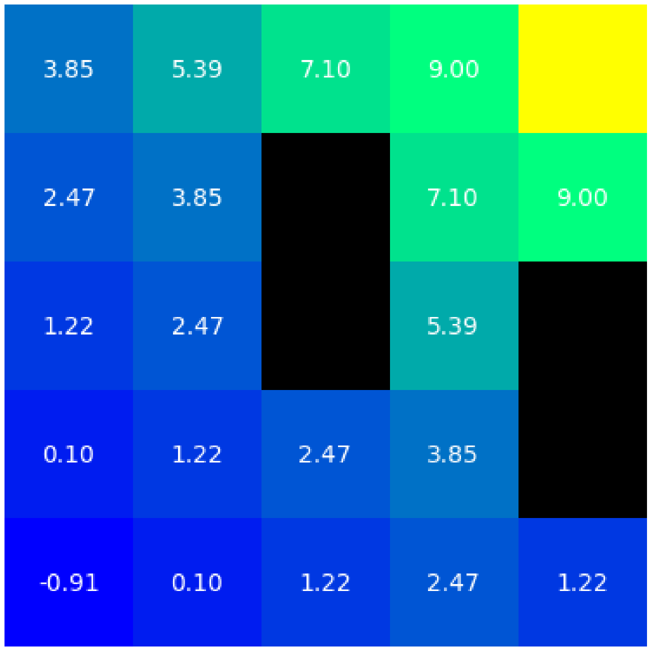

Fig. 11. Grid World after iteration 5

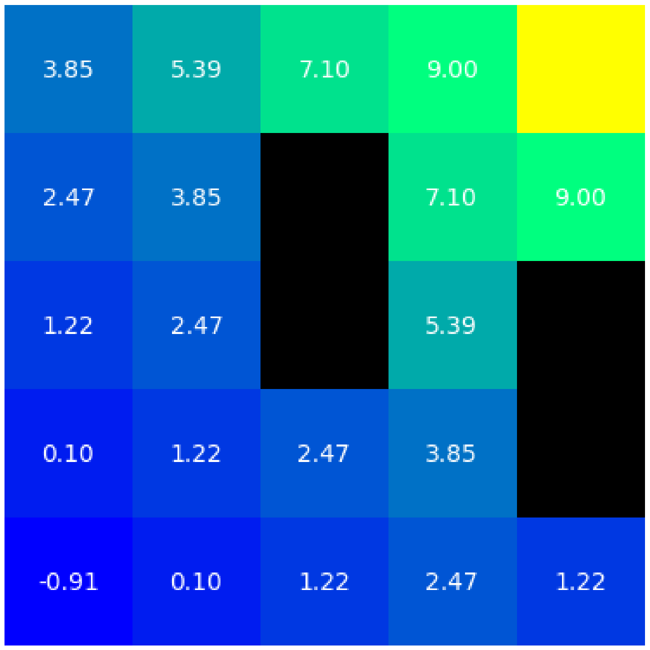

Fig. 12. Converged Grid World rewards at each state

D. *Discussion of Observations*

After the first iteration, the majority of the left three columns of the grid world only have the step reward of -1 and have not yet fully explored the actions to reach the terminal state. From the reward values of the two columns on right side, the cells closest in proximity to the terminal state have the highest rewards and are closest to the reward of +10, which was defined as the goal state, and decrease as the states become further away.

The second iteration results in the middle column having positive reward values that correspond to their proximity in movements to the terminal state.

The third iteration results in the 2nd to the left column having positive reward values and the fourth iteration results in the left most column having its reward values. Though for that column, not all rewards are positive since the cell furthest away from the terminal state is negative because of the step reward of -1 and the discount factor of $\gamma = 0.9$.

There is a fifth iteration with no discernible difference from the fourth iteration due to the small threshold for convergence, $\theta = 1 \times 10^{-10}$.

The converged grid world rewards at each state is the same as the final two iterations from the Results section. As states progressively get further away from the terminal state, the reward value for those states keep decreasing.

The converged grid world actions at each state provides another insightful look into the grid world scenario. Instead of showing the rewards, the best actions to take to reach the terminal state are labeled. This plot is particularly interesting as it's a nice depiction showing the states navigating around the blocked cells to reach the terminal state.

Lastly, we have the value function changes plotted over the iterations of the algorithm. We found the maximum difference between any state from one iteration to the next and plotted that over the iterations. The steep drop from iteration 4 to 5 exists because the value function change must be $\leq \theta$, the convergence threshold, to exit the algorithm, so an additional iteration occurs to satisfy that requirement.

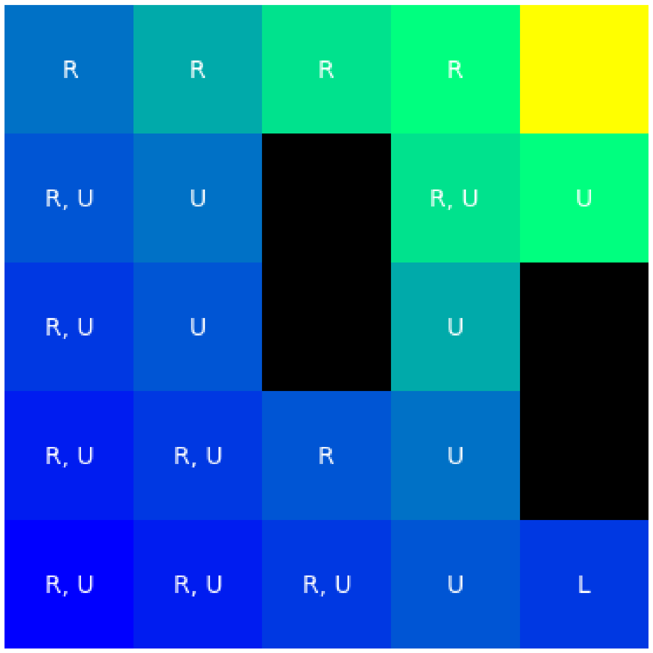

Fig. 13. Converged Grid World actions at each state

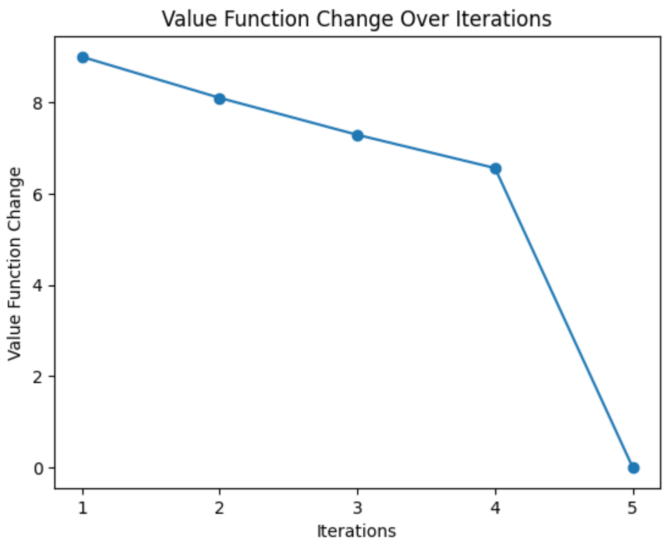

Fig. 14. Value Function Change vs Iterations of the value iteration algorithm for Grid World

E. Extra Discussion on Choice of parameters

This section was added on at the end for some brief discussion on how the choice of various parameters (including relative reward values for the goal and blocked states along with the step reward and discount factor γ). Figure 15 below shows a difference convergence plot for the following set of parameters: 0 for the goal state, -100 for the blocked states, -1 for the reward step, and $\gamma = 0.9$. All other parameters were kept the same (e.g., grid size and goal and blocked states placement). If we remember the parameters we used previously above, the same reward step and gamma were used. However, the main thing to notice is the greatly reduced goal state reward (from +10 to 0) and greatly reduced blocked state reward (from 0 to -100, or effectively some negative number with a large magnitdue). The purpose of these particular changes was to see if the number of iterations could be elongated from the relatively fast convergence we observed above. Previously, the iteration number was 5 (starting from iteration 1) was convergence was reached. However, for the new set of changes, the iteration number was 8 (starting from iteration 0, or iteration 9 if the offset is accounted for starting at 1 instead) – almost double the required number of iterations to reach convergence.

Comparing the convergence pattern observed, we see that the effect of a $\gamma < 1$ is more apparent where the value function change at each function seems to decrease by the same value given by γ. The rate of decrease of the value function change should then depend on γ. However, a lower discount factor might affect the resulting policy – the θ threshold might be reached sooner but depending on the reward choices for the goal and blocked states along with the reward step, the incorrect policy might be outputted from the program. This was seen with relatively low magnitude reward for the blocked state (e.g., -1) where hitting an obstacle did not provide a big enough penalty to completely rule out that action when determining the best actions at each state. Regardless, the γ produced a clear dampening effect on the value function change with varying effects depending on the reward choices and may affect how fast convergence is achieved.

Lastly, the magnitudes of the reward for the goal state and reward step are relatively close compared to the previous parameters above (i.e., +10 and -1 compared to 0 and -1, respectively). This also affected convergence where more iterations of the value iteration was required to gauge how important it was to reach the goal state – unlike the previous plots above where a decent portion of the cells finalized their value function values from the first iteration, the plots with these new parameters did not have any such cells since the goal essentially gave nothing to the value estimate (reward of 0 when reaching the goal). So, more iterations were required as other cells had their value estimate decrease until a way to the goal was found from their neighbors. This produced a slower wave of updates as the ”correct” action took longer to find because the goal did not make as much of an impact as above with a higher reward when reaching it.

For the other plots (grids) generated, the same policy was produced (as expected) and the coloring of the value iteration grid was similar despite having different values in each cell – emphasizes the importance of relative changes and relative reward scaling rather than absolute changes.

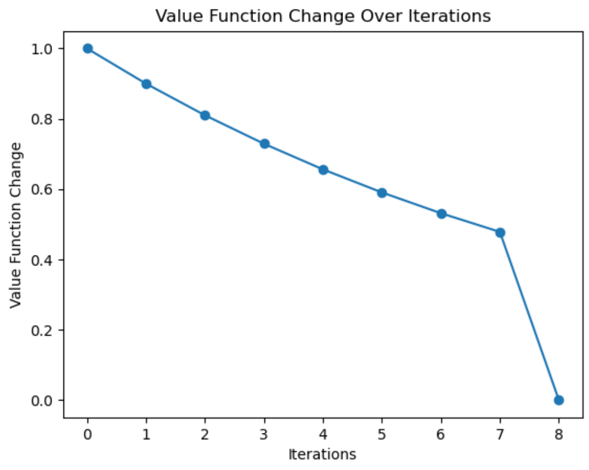

Fig. 15. Value Function Change vs Iterations of the value iteration algorithm for Grid World